CS350: Assignment 1 – Loading Programs

Tavian Barnes, Emil Tsalapatis

In Assignment 0, you set up CastorOS to build successfully, but the boot process did not complete successfully. In this assignment, you will implement the remaining boot functionality. The missing functionality is part of the program loader that reads executable files, copies them into memory, and passes command line arguments. Once implemented, you will be able to run simple commands like `ls` and `cat`.

Important: CastorOS has been updated to fix some critical bugs. Before completing this assignment, download and extract the latest copy of the source code:

```
$ python client.py download
$ tar zxvf castoros-latest.tar.gz --exclude sys/dev/console.c
```

The `--exclude` option prevents it from overwriting your Assignment 0 solution. If you omit it, you will have to re-do Assignment 0 (without submitting it) before continuing with Assignment 1.

1 Completing the Loader

Executable files contain a mixture of code and data. To run a program, the code and data must be loaded from disk to memory. This is the responsibility of a *program loader*. There are a few popular formats for executables, one of which is ELF [2] (Executable and Linkable Format), used by CastorOS, Linux, and BSD.

The loader does not copy an ELF binary verbatim into memory. Instead, the *header* at the beginning of the file tells the loader which *segments* to load. Therefore, to load an ELF program, the loader must read and interpret the header. This code is incomplete in CastorOS, so when it tries to run `/sbin/init` during boot, it crashes:

```
$ qemu-system-x86_64 -smp cpus=1 -kernel build/sys/castor \
    -hda build/bootdisk.img -nic none -nographic
SeaBIOS (version rel-1.16.3-0-ga6ed6b701f0a-prebuilt.qemu.org)
Booting from ROM..Castor Operating System
...
loader: Jumping to userspace
CPU 0
Interrupt 14 Error Code: 0000000000000004
...
Entered Debugger!
kdbg>
```

Tip: Press `Ctrl+A`, `X` to exit QEMU.

1.1 Reading the Header

The first step is to read the ELF header from disk. Line 127 of `sys/kern/syscall.c` currently says

```
/* XXXFILLMEIN: Load the ELF headers into the page. */
```

You must fill in this code to look up the path, open the file, and read the first 1024 bytes. The VFS (Virtual File System) APIs from `sys/include/vfs.h` are used to access the file system. Some of the relevant APIs are:

```
// Looks up a path in the file system.
// Returns a VNode on success, or NULL on failure.
VNode *VFS_Lookup(const char *path);

// Opens a file that has already been looked up.
// Returns 0 on success, -1 on failure.
int VFS_Open(VNode *fn);

// Reads ‘len‘ bytes, starting from offset ‘off‘, from an open
// file into the buffer ‘buf‘.
// Returns the number of bytes read.
int VFS_Read(VNode *fn, void *buf, uint64_t off, uint64_t len);
```

Use these functions to look up the `path`, open the `file`, and read the first 1024 bytes into the page of memory `pg`. Unlike userspace, file handles inside the kernel are represented by the VNode object. You will need to look up and open the file (VNode).

1.2 Loading the Segments

In this part of the assignment we need to read the ELF header, then read the program header table, and finally load each segment. The ELF header is in the first 1024 bytes of the file (starting at 0), and you will use it to find where the program headers are. Parsing the program headers will allow you to load the segments.

The format of the ELF header is described by this C structure (from `sys/include/elf64.h`):

```
typedef struct {
       unsigned char  e_ident[EI_NIDENT];   /* File identification. */
       Elf64_Half     e_type;       /* File type. */
       Elf64_Half     e_machine;    /* Machine architecture. */
       Elf64_Word     e_version;    /* ELF format version. */
       Elf64_Addr     e_entry;      /* Entry point. */
       Elf64_Off      e_phoff;      /* Program header file offset. */
       Elf64_Off      e_shoff;      /* Section header file offset. */
       Elf64_Word     e_flags;      /* Architecture-specific flags. */
       Elf64_Half     e_ehsize;     /* Size of ELF header in bytes. */
       Elf64_Half     e_phentsize;  /* Size of program header entry. */
       Elf64_Half     e_phnum;      /* # of program header entries. */
       Elf64_Half     e_shentsize;  /* Size of section header entry. */
       Elf64_Half     e_shnum;      /* # of section header entries. */
       Elf64_Half     e_shstrndx;   /* Section name strings section. */
} Elf64_Ehdr;
```

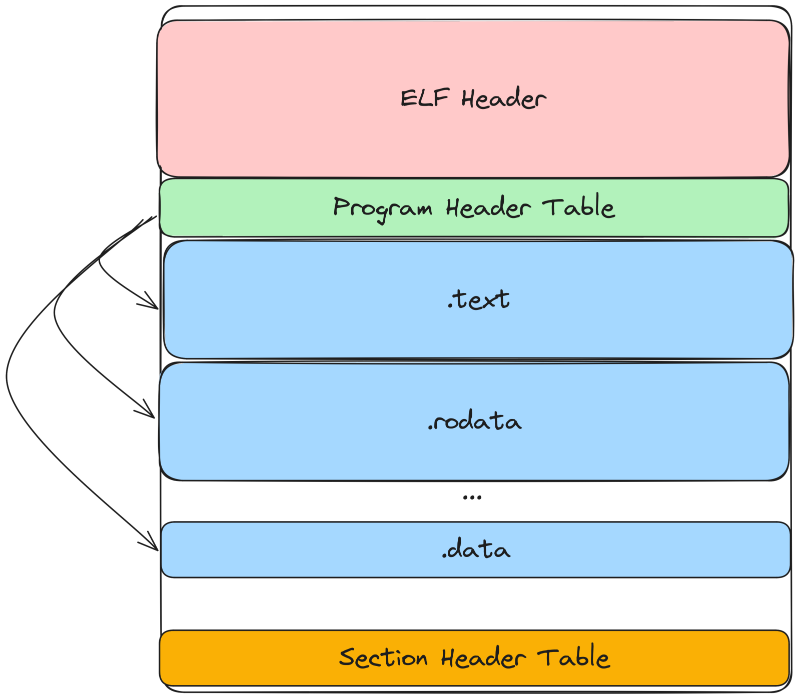

Figure 1: The ELF file header.

> **Tip:** You can examine the ELF header of a binary with the `readelf` command:
>
> `$ readelf --file-header build/sbin/init`

We are specifically interested in the *program headers*. `e_phoff` tells us where they are within the file, and `e_phnum` tells us how many of them there are. Each program header looks like this:

```
typedef struct {
        Elf64_Word    p_type;       /* Entry type. */
        Elf64_Word    p_flags;      /* Access permission flags. */
        Elf64_Off     p_offset;     /* File offset of contents. */
        Elf64_Addr    p_vaddr;      /* Virtual address in memory image. */
        Elf64_Addr    p_paddr;      /* Physical address (not used). */
        Elf64_Xword   p_filesz;     /* Size of contents in file. */
        Elf64_Xword   p_memsz;      /* Size of contents in memory. */
        Elf64_Xword   p_align;      /* Alignment in memory and file. */
} Elf64_Phdr;
```

> **Tip:** You can also use `readelf` to view the program headers:
>
> `$ readelf --program-headers build/sbin/init`

Program headers with `p_type == PT_LOAD` specify a segment of the file to load into memory. `p_offset` tells us where the segment is located in the file, and `p_filesz` tells us its size on disk. `p_vaddr` tells us where the segment should be loaded in memory, and `p_memsz` tells us its size in memory. If `p_filesz < p_memsz`, the loader should read `p_filesz` bytes, and set the remaining bytes to zero.

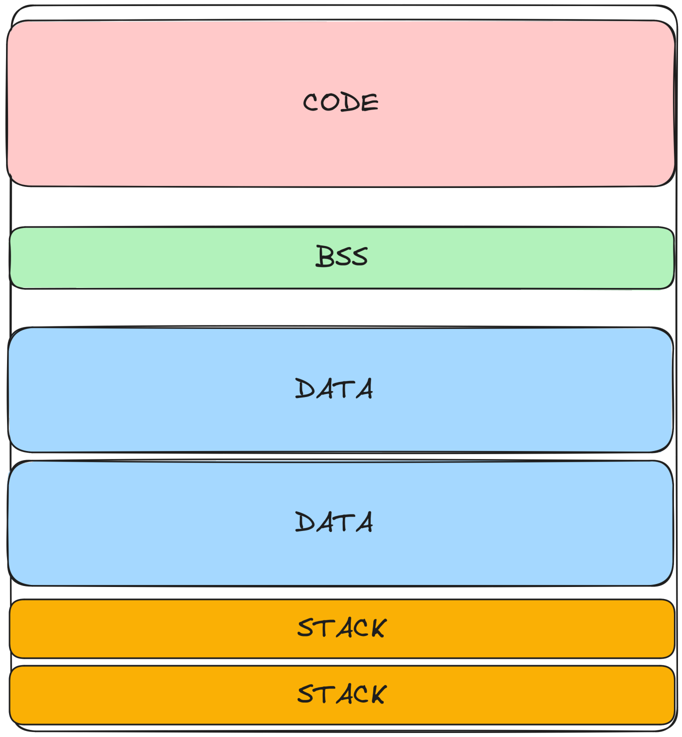

Figure 2: The memory layout of a userspace program. The code segment holds the executable program code. The BSS holds prezeroed variables, while the data segments are empty space created at runtime by the program itself. The kernel places the stack of each thread in the process in a dynamically created stack segment.

The code to load all segments belongs at line 171 of `sys/kern/loader.c`, which currently says:

```
/* XXXFILLMEIN: Load the ELF segments. */
```

Replace this with a loop over the `phdr` array. For each `PT_LOAD` program header, load the segment from the file, and zero the remaining bytes. These helper functions are available for you:

```
void LoaderLoadSegment(AS *as, VNode *vn, uintptr_t vaddr,
                       uintptr_t offset, uintptr_t len);
void LoaderZeroSegment(AS *as, uintptr_t vaddr, uintptr_t len);
```

`LoaderLoadSegment` will load data from the VNode to a specified virtual address, given a file offset and length that you want to read in. `LoaderZeroSegment` will zero a range of memory from vaddr to length.

> **Tip:** If you are having trouble computing the right offsets, lengths and addresses, this is a good opportunity for kprintf debugging (see Section 4). Print and compare the values you computed with the values from readelf to make sure everything makes sense.

2 Passing Arguments

CastorOS doesn't implement `fork`, and instead creates new processes with the `spawn` system call. Spawn combines `fork` and `exec` from UNIX-like operating systems into a single call. For example, a shell will spawn a new process for each command. The user types a command in the command line, and the shell responds by passing the arguments over to `spawn`. The `spawn` call creates, loads and runs a new process, and the shell waits for it to exit before returning control to the user.

Let's take as an example an invocation of the `cat` shell command. The command opens all input files and prints their contents to the terminal. If file `a.txt` has contents `"Hello\n"` and file `b.txt` has contents `"World\n"`, then:

```
$ cat a.txt b.txt
Hello
World
```

The shell receives the arguments `[cat, a.txt, b.txt]`. The shell then resolves the full path of `cat` to `/bin/cat` and calls `spawn`:

```c
char *path = "/bin/cat";
char *args = { "cat", "a.txt", "b.txt" };
spawn(path, args);
```

The current `spawn` implementation copies the arguments into kernel space, but it does not copy them back out to the new userspace process. Therefore, any shell command that uses arguments will not work.

For this part you will fill in line 158 of `sys/kern/syscall.c` that currently says:

```c
/* XXXFILLMEIN: Export the argument array out to the new application. */
```

To finish the assignment, write the code to copy the argument strings into the new process's address space, so that they can be passed to the `main` function. Figure 3 shows an example of the desired memory layout. The system call assumes there are up to 7 arguments and that each argument is at most 256 bytes long, so they fit in the 4 KiB buffer allocated in `Spawn`.

You can find pointers to the argument strings in the `arg` variable. The `argstart` variable points to the destination buffer. It may help to cast these variables to the appropriate types:

```c
char **in_argv = (char **)arg + 1;
uintptr_t *out_argc = (uintptr_t *)argstart;
uintptr_t *out_argv = out_argc + 1;
char *out_args = (char *)(out_argv + 7);
```

Now, use a loop to copy the strings from `in_argv` to the output. Once you see `in_argv[i] == NULL`, you have reached the end of the argument list. At that point, set `*out_argc` to the number of arguments.

Tip: Use `strcpy` or `memcpy` to copy `in_argv[i]` to `out_args`, then increment `out_args` by the number of bytes you copied.

Warning: You cannot simply write `out_argv[i] = (uintptr_t)out_args` to fill in `out_argv`, because `out_args` is a kernel pointer. Instead, calculate the equivalent userspace pointer. Look at the definition of `argstart` to find the base userspace address.

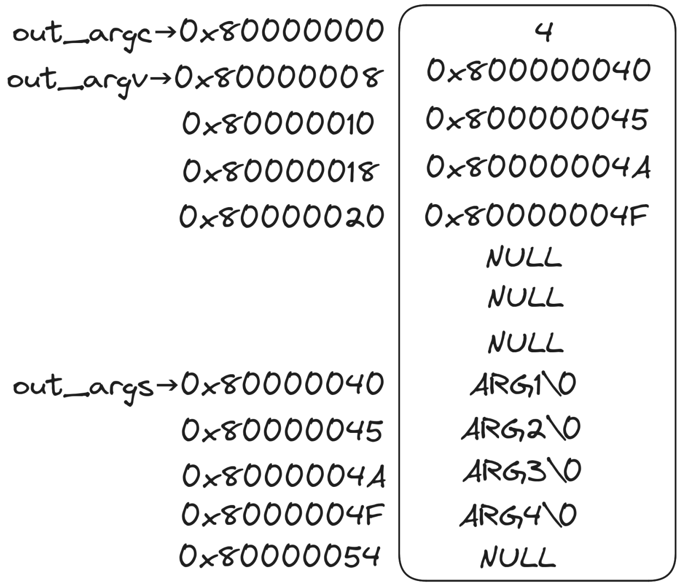

Figure 3: The layout of the arguments page in userspace. The first 8 words (64 bytes) are the argument array. The first entry holds the number of entries in the array, the rest hold either the userspace address of an argument or `NULL`. We place the argument strings themselves directly after the array. In this case our arguments are the four strings form `arg1` to `arg4` along with the string termination character `\0`. The strings are have size 5 and are stored consecutively in the page. The pointers in the array come in multiples of 5 for this reason.

Once you finish this assignment, upon booting CastorOS you should see a shell prompt. Try running some commands to check that argument passing is working, for example:

```
Shell> echo hello world
echo hello world
Shell> ls /
boot
dev
bin
sbin
tests
LICENSE
Shell> cat /LICENSE
Copyright (c) 2013-2023 Ali Mashtizadeh

...
```

Unlike UNIX, our `ls` requires a single argument for the absolute path you wish to list.

3 Submitting Your Solutions

3.1 Submitting Results

Submitting Assignment 1 works just like Assignment 0. Use `git commit` to create a single commit with your Assignment 1 solution, (on top of your previous commit with your Assignment 0 solution). **The commit should only include sys/kern/syscall.c and sys/kern/loader.c. If the commit includes any other files the server will automatically reject the submission.** Next, follow these steps to generate and submit your Assignment 1 patch.

```
$ python client.py patch
$ python client.py submit -a asst1
```

The status of your submission can be monitored using the `client.py status` command:

```
$ python client.py status -a asst1
TOTAL: 9/9
Evaluated at 26/09/2024 12:00
=========END OF SUBMISSION=========
```

4 Debugging CastorOS

For debugging we use three tools: `kprintf` debugging, the `kdbg` kernel debugger from inside the OS, and the `GDB` from outside the OS. You should select the tool based on the nature of the bug we are triaging.

4.1 Print Debugging

The simplest technique at our disposal is `kprintf` debugging. We add `kprintf` statements in the kernel to print helpful diagnostics throughout execution. Using `kprintf` is quick and easy but requires changing the code every time we want to move or change a print message. Using `kprintf` is also not always possible, e.g., during early boot.

Make sure to remove all `kprintf` messages from your code before you submit. The grading scripts used to evaluate submissions read the serial console of the QEMU machine to check whether basic shell commands like `ls` and `cat` work correctly. Leftover `kprintf` messages may write to the console when these commands are being executed and cause the submission to fail the script's tests even if it is correct.

4.2 Builtin Kernel Debugger (kdbg)

You can also use `kdbg`, CastorOS's builtin kernel debugger. The main advantage of `kdbg` is its direct access to kernel state. CastorOS enters `kdbg` either if there is a kernel crash or if we run the `bkpt` command from the CastorOS terminal. If we enter `kdbg` using `bkpt` we resume execution with `continue` command.

Running `help` in `kdbg` gives us a list of commands we can run. Each command gives us information about a certain part of the system, e.g. CPU state or running processes/threads. Using `kdbg` is ideal for bugs that do not crash the system immediately but eventually cause problems or crashes.

4.3 GDB

Another tool at our disposal is the standard GDB debugger. QEMU allows GDB to directly attach and debug the OS as if it was a regular process. To use GDB with CastorOS we first add the `-s` and `-S` flags when invoking QEMU [1]. These flags will make QEMU listen for connections from a local GDB instance and also prevent it from running the OS immediately.

If you want debug symbols you will need to change the BUILDTYPE to DEBUG by editing your Local.sc and adding BUILDTYPE="DEBUG" to it. I would recommend removing this when you don't need it.

```
qemu-system-x86_64 -s -S -nic none -m 64 -smp cpus=1 -nographic \
    -kernel build/sys/castor -hda build/bootdisk.img
```

We use GDB by running the following command from another terminal:

```
(gdb) target remote localhost:1234
```

This command will attach the GDB instance into CastorOS running inside QEMU. Running `continue` will allow CastorOS to start booting.

```
(gdb) continue
```

Please refer to one of the many GDB tutorials out there for details on how to use GDB for debugging.

References

[1] GDB usage. https://qemu-project.gitlab.io/qemu/system/gdb.html, August 2023.

[2] The ELF File Format. https://wiki.osdev.org/ELF, August 2023.